Нижегородский государственный университет им. Н.И. Лобачевского
Институт информационных технологий, математики и механики

Компиляторы для глубоких нейросетевых моделей

Обзор Apache TVM

При поддержке компании YADRO

Родимков Ю.А.,
Кустикова В.Д.

Содержание

ВВЕДЕНИЕ

Apache TVM

- ***Apache TVM (Tensor Virtual Machine)*** – тензорный компилятор с открытым исходным кодом, который специализируется на *выполнении нейронных сетей* на CPUs, GPUs и ускорителях машинного обучения
- ***Основные возможности:***
  - *Компиляция глубоких моделей* в исполняемые на устройстве модули
  - Предоставление инфраструктуры для *автоматического развертывания и оптимизации моделей* на большом количестве бекэндов*

- *Примечание:* поддерживает кросс-компиляцию исполняемого модуля, удаленный запуск (через Remote Procedure Call, RPC), обучение нейронных сетей и квантизацию обученных моделей

* Бекэнд – программно-аппаратное окружение.

Цель разработки Apache TVM

- ***Цель –*** возможность оптимизировать и эффективно выполнять вычисления на любом аппаратном обеспечении

- ***Ключевые особенности:***
  - *Переносимость*. Возможность переноса модели между разным типом устройств
  - *Эффективность*. Эффективное использование целевой платформы
  - *Поддержка новых аппаратных платформ*. Возможность добавлять специализации применяемых оптимизаций для собственной аппаратной платформы и интегрировать сторонние средства кодогенерации под эту платформу

Примеры использования

- При каждом включении ***Alexa*** на всех устройствах используется модель, оптимизированная с помощью TVM

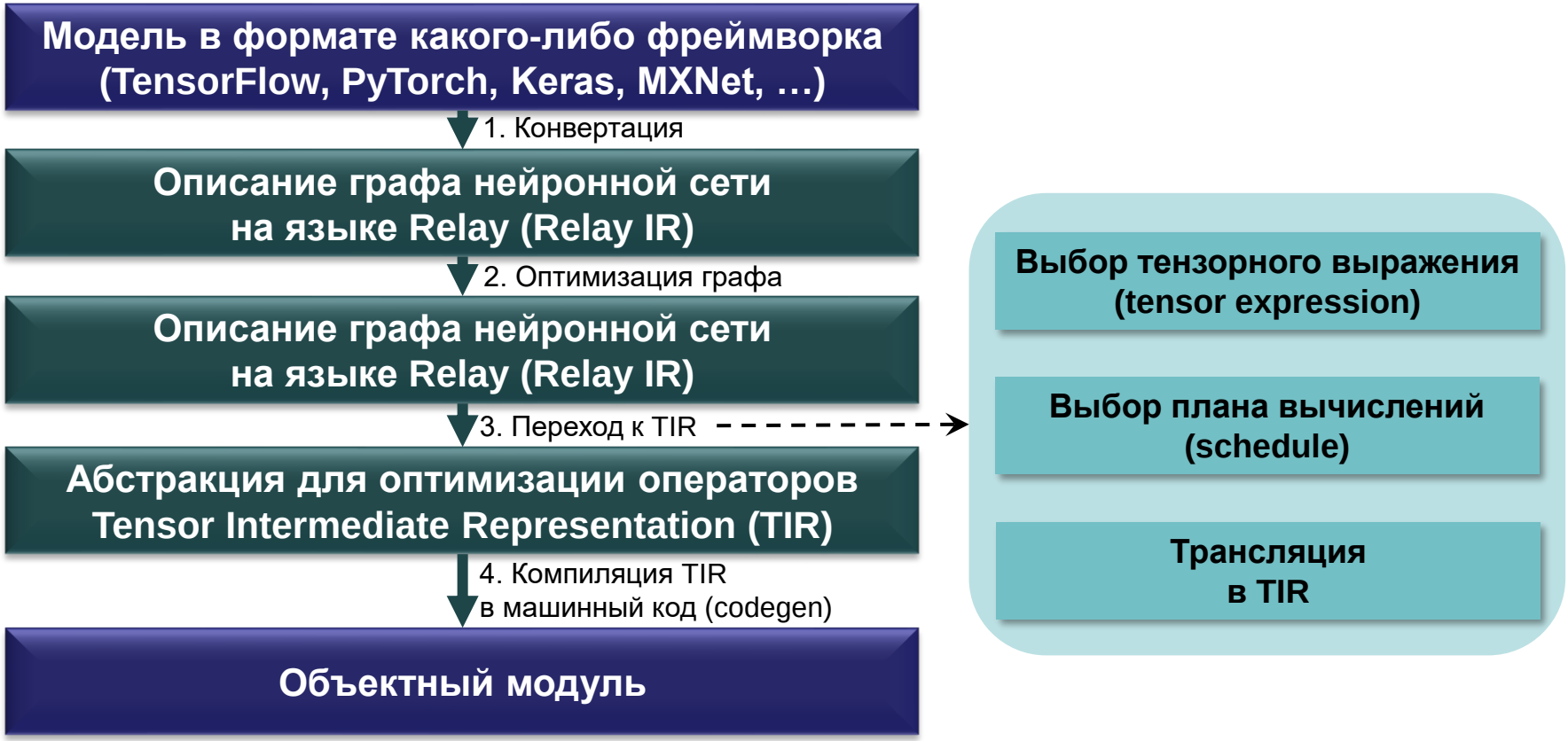
- ***Microsoft Bing query***: 112 мс (TensorFlow) >> 34 мс (TVM)
- ***QnA-бот***: 73 мс >> 28 мс (CPU), 10,1 мс >> 5,5 мс

- «TVM – ключ для доступа к ML в ***Hexagon***»

* Apache TVM tutorials [https://github.com/Deelvin/apache-tvm-tutorials].

ПОСЛЕДОВАТЕЛЬНОСТЬ ОПТИМИЗАЦИИ НЕЙРОННЫХ СЕТЕЙ

Типовая схема работы Apache TVM



Конвертация модели в Relay IR

- Relay – функциональный язык для представления графа нейронной сети

- ***Relay IR*** – промежуточное представление, в которое конвертируется модель после загрузки
- ***Relay IR*** – абстракция для хранения модели
  - Обеспечивает независимый от исходного фреймворка способ представления нейронной сети

- Поддерживаемые форматы моделей: MXNet, OneFlow, ONNX, TensorFlow, TensorFlow Lite (TFLite), Keras, Core ML, Caffe, Caffe2, Darknet, PyTorch, PaddlePaddle

Оптимизация графа вычислений

- К графу нейронной сети в формате Relay IR применяются ***проходы*** (Pass)
- Проход = трансформация графа
- Примеры автоматических трансформаций:
  - Преобразование размещения структур данных (layout transform)
  - Слияние операторов (operator fusion)
  - Замена операторов на более оптимизированные (могут выполняться вычисления с меньшей точностью)
  - Свертка констант (constant-folding)
  - Объединение нескольких параллельных сверточных или полносвязных слоев

Переход к TIR. Тензорное выражение

- В TVM вводится ***язык тензорных выражений (tensor expression)*** для последующей автоматической генерации кода
- Каждая операция в графе описывается размером выходного тензора и формулой, которая определяет, каким образом вычислять его элементы

```
A = te.placeholder((128,), name="A")                    # пустой тензор
B = te.placeholder((128,), name="B")                    # пустой тензор
C = te.compute(A.shape, lambda i: A[i] + B[i], "C")     # сложение векторов
```

- ***Тензорное выражение*** описывает аппаратно-независимую реализацию операции
- Доступные примитивы: объявление переменных, объявление и проход по элементам тензоров, объявление и итерирование по осям редукции (например, в матричном умножении ось редукции соответствует циклу скалярного произведения строки на столбец)

Переход к TIR. План выполнения

- ***План (schedule)*** – набор преобразований, применяемых к тензорному выражению
- Строится посредством применения преобразований, сохраняющих логическую эквивалентность программы
  - Описывает программные оптимизации, такие как развертывание, слияние, разбиение на блоки, векторизация, распараллеливание, привязка потоков и т.д.

```
s = te.create_schedule(C.op)           # создание пустого плана вычислений
yo, yi = s[C].split(C.op.axis[0], 16) # разделение оси на линии по 16
s[C].parallel(yo)                      # распараллеливание внешнего цикла
```

- TVM позволяет автоматизировать выбор плана выполнения из заранее описанных или сгенерированных планов
- ***Тензорное выражение обеспечивает математическое описание алгоритма, а план описывает, как нужно выполнять вычисления***

Переход к TIR. Тензорное внутреннее представление

- ***Тензорное внутреннее представление (Tensor Intermediate Representation, TIR)*** – низкоуровневое представление для операций над тензорами
- TIR формируется для операторов в графе вычислений на основании соответствующих тензорных выражений и планов исполнения
- ***TIR – абстракция для автоматической оптимизации программ***
- Пример TIR для умножения векторов с планом по умолчанию:

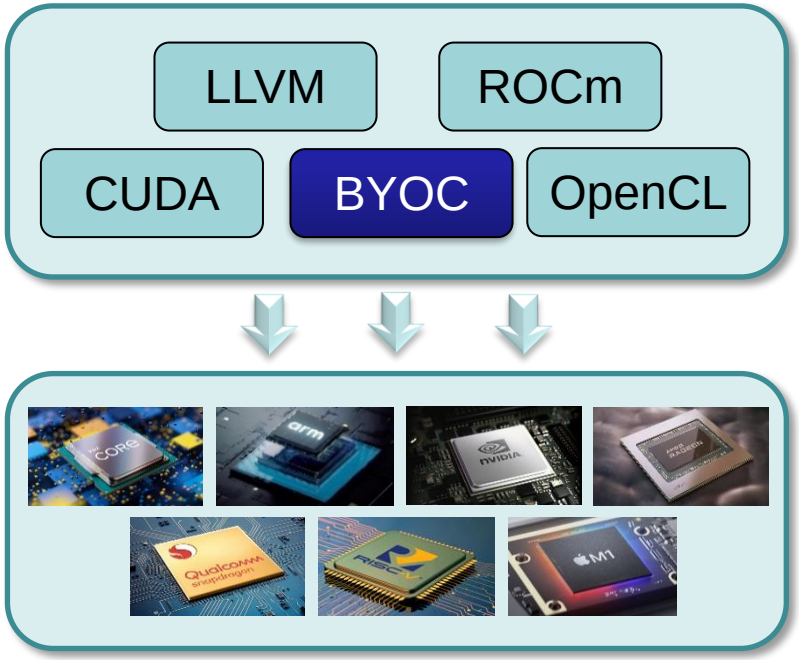
```
@I.ir_module
class Module:
    @T.prim_func
    def main(
        A: T.Buffer((128,), "float32"),
        B: T.Buffer((128,), "float32"),
        C: T.Buffer((128,), "float32")):
        for i in range(128):
            C[i] = A[i] + B[i]
```

* Feng S., et al. Tensorir: An abstraction for automatic tensorized program optimization // Proc. of the 28th ACM Int. Conf. on Arch. Support for Prog. Languages and Operating Systems, Vol. 2. – 2023. – P. 804-817.

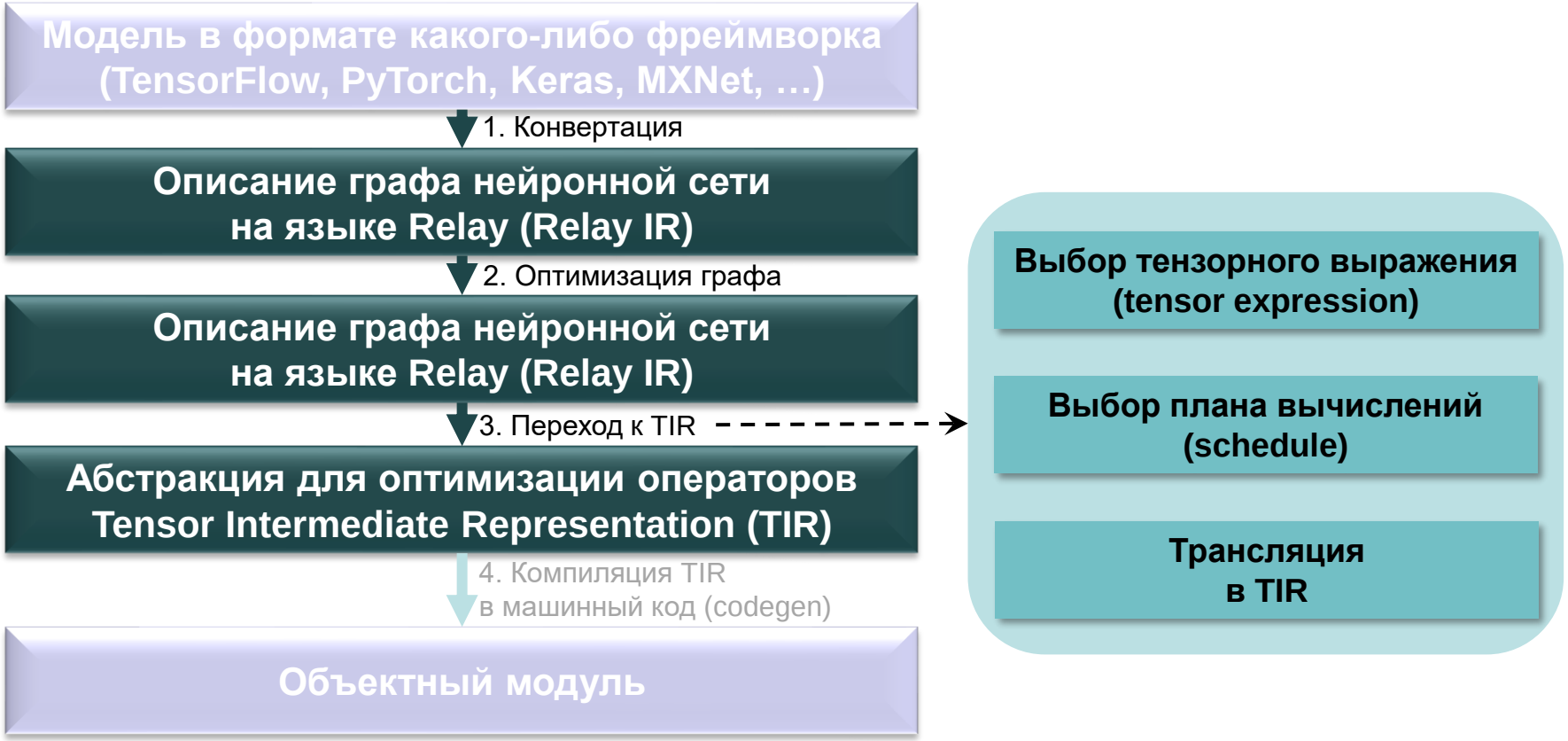
Генерация машинного кода

- ***Генерация машинного кода (codegen)*** на основании TIR выполняется средствами существующих компиляторов

- Поддерживаемые бэкенды компиляторов:
  - LLVM для x86, ARM, AMDGPU и NVPTX, а также любая другая платформа, поддерживаемая LLVM
  - Специализированные компиляторы (например, NVCC от NVIDIA)
  - Собственные бэкенды, которые можно реализовать с помощью TVM («Bring Your Own Codegen», BYOC)

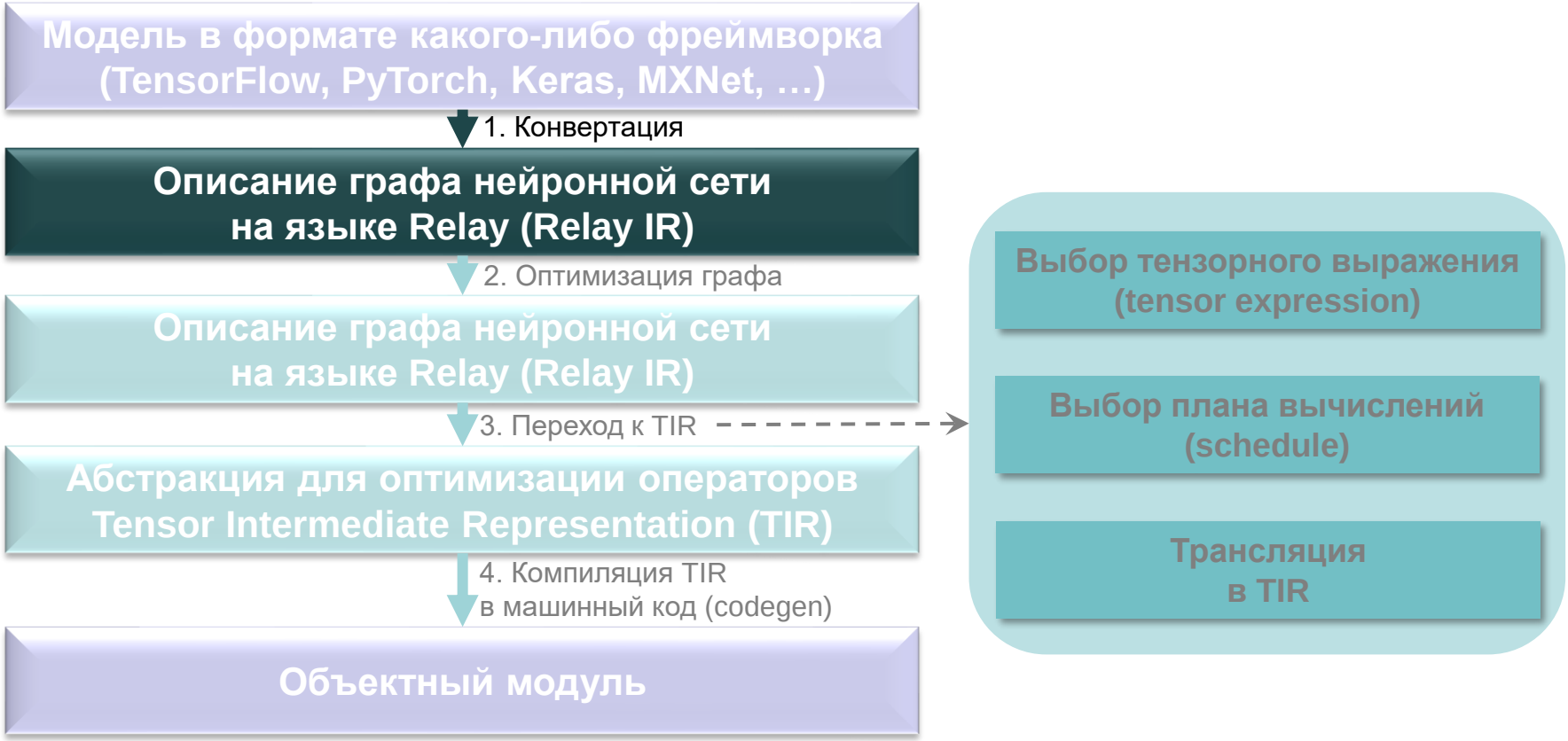


Типовая схема работы Apache TVM. Подробнее в лекции



КОНВЕРТАЦИЯ МОДЕЛИ В ГРАФ ВЫЧИСЛЕНИЙ

Типовая схема работы Apache TVM



Конвертация модели в Relay IR

- Relay – функциональный язык для представления графа нейронной сети

- ***Relay IR*** – промежуточное представление сети в виде графа зависимостей с использованием высокоуровневых абстракций слоев, в которое конвертируется модель после загрузки
  - Обеспечивает независимый от исходного фреймворка способ представления нейронной сети
  - Содержит простые (сложение, умножение тензоров) и высокоуровневые примитивы (слои нейронной сети)

* Roesch J., et al. Relay: A new ir for machine learning frameworks // Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. – 2018. – P. 58-68.

Слои нейронной сети

- Relay IR описывает нейронную сеть как ***граф зависимостей*** на уровне слоев нейронной сети ***без указания конкретной реализации операторов***
- Relay содержит различные классические операторы для нейронных сетей:
  - Умножение и сложение с константой, сложение тензоров
  - Слои функций активаций
  - Полносвязный слой (fully connected layer)
  - Сверточный слой (convolutional layer)
  - Слой транспонированной свертки (transposed convolution)
  - Различные вариации пулинга (pooling)
  - Нормализация по пачке (batch normalization) и по слою (layer normalization)
- На данный момент готовится второе поколение Relay IR – Relax* (Relay Next)

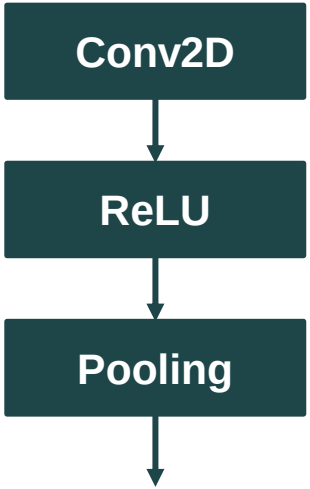
* Lai R., et al. Relax: Composable Abstractions for End-to-End Dynamic Machine Learning // arXiv preprint arXiv:2311.02103. – 2023.

Пример реализации нейронной сети

- Пример реализации нейронной сети на Relay:

```
input = tvm.relay.var("input", shape=input_shape, dtype='float32')
weight = tvm.relay.var("weight", shape=kernel_shape, dtype='float32')

conv = relay.nn.conv2d(
  input, weight, padding=[1, 1, 1, 1], channels=kernel_shape[-1],
  kernel_size=[3, 3], data_layout="NHWC", kernel_layout="HWIO"
)
conv = relay.nn.relu(conv)
conv = relay.nn.avg_pool2d(conv)
```

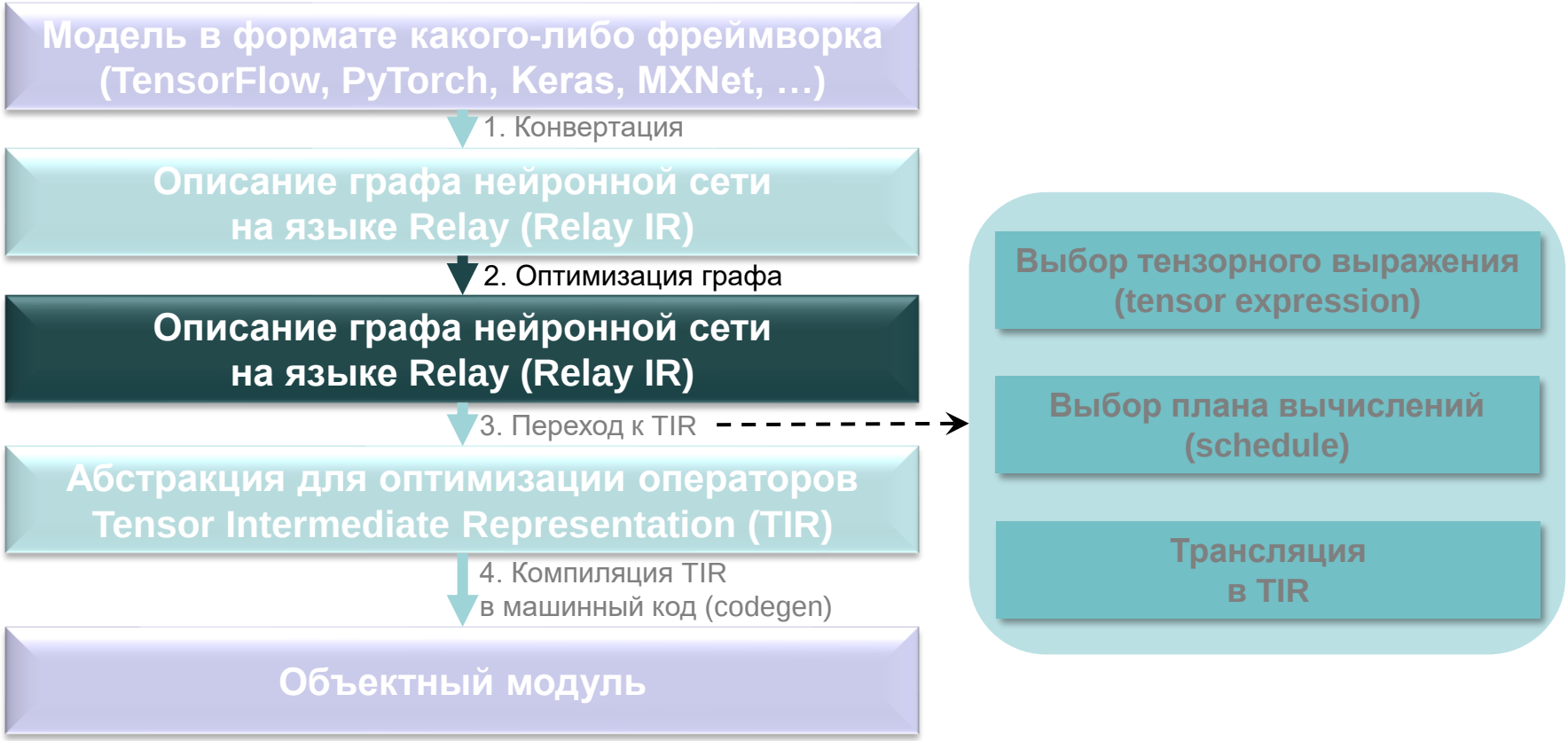


- Результат вывода объекта типа **`IRModule`**:

```
fn (%input: Tensor[(1, 26, 26, 32), float32], %weight: Tensor[(3, 3, 32, 32), float32]) {
  %0 = nn.conv2d(%input, %weight, padding=[1, 1, 1, 1], channels=32, kernel_size=[3, 3],
  data_layout="NHWC", kernel_layout="HWIO");
  %1 = nn.relu(%0);
  nn.avg_pool2d(%1, pool_size=[1, 1], padding=[0, 0, 0, 0])
}
```

ОПТИМИЗАЦИЯ ГРАФА ВЫЧИСЛЕНИЙ НЕЙРОННОЙ СЕТИ

Типовая схема работы Apache TVM



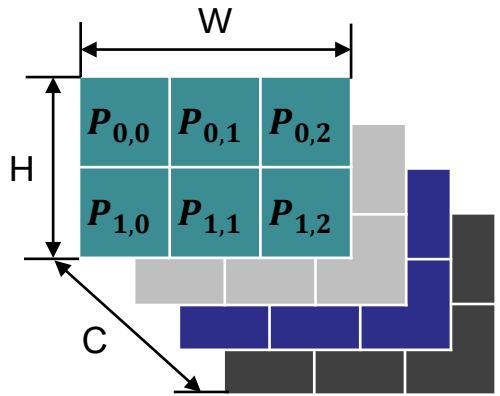
Оптимизации графа нейронной сети

- К графу нейронной сети в формате Relay IR применяются ***проходы (Pass)***
- Проход = трансформация графа
- Примеры автоматических трансформаций:
  - *Преобразование размещения структур данных (layout transform)*
  - *Слияние операторов* (operator fusion)
  - *Замена операторов на более оптимизированные (могут выполняться вычисления с меньшей точностью)*
  - Свертка констант (constant-folding)
  - Объединение нескольких параллельных сверточных или полносвязных слоев

- *Примечание:* поддержка слоев квантованных моделей обеспечивается с использованием проходов на уровне Relay IR

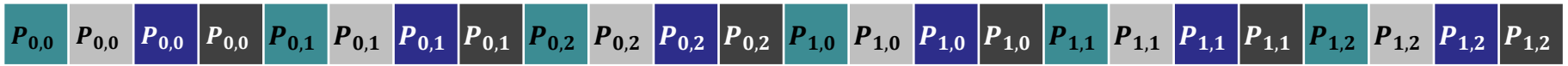
Преобразование размещения структур данных…

- ***Преобразование размещения структур данных (layout transform)*** – оптимизация размещения данных
- ***Цель преобразования*** – улучшение работы с памятью при обходе данных
- Основная операция для преобразования – свертка
- Используемые обозначения для описания порядка хранения данных:
  - N – количество объектов данных
  - H, W – высота и ширина тензора входного объекта
  - C – количество каналов в тензоре входного объекта
  - I, O – количество входных (input) и выходных (output) каналов в ядре свертки
  - c, i, o – факторы группировки каналов тензора входного объекта, входных и выходных каналов в ядре свертки соответственно

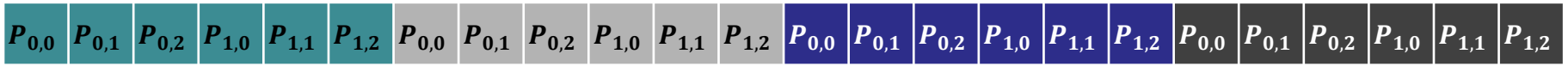


Преобразование размещения структур данных…

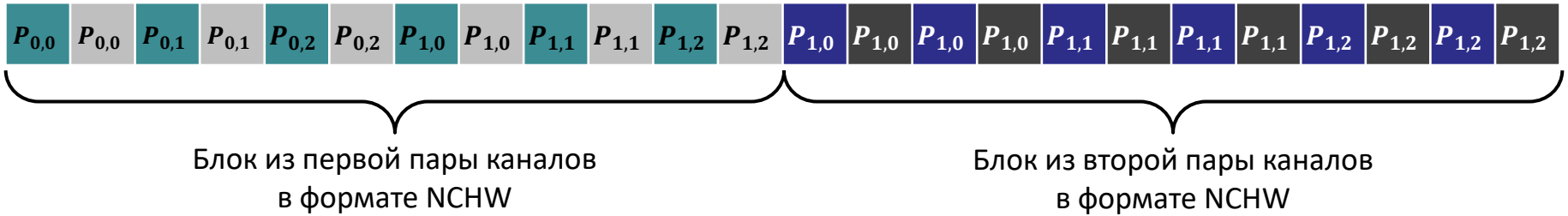
- Форматы хранения данных и ядер свертки:
  - Данные NHWC и ядро HWIO (Keras и TensorFlow)



  - Данные NCHW и ядро OIHW (PyTorch)



  - Данные NCHWc и ядро OIHWio (инструменты для вывода). Пример NCHWc (c=2):

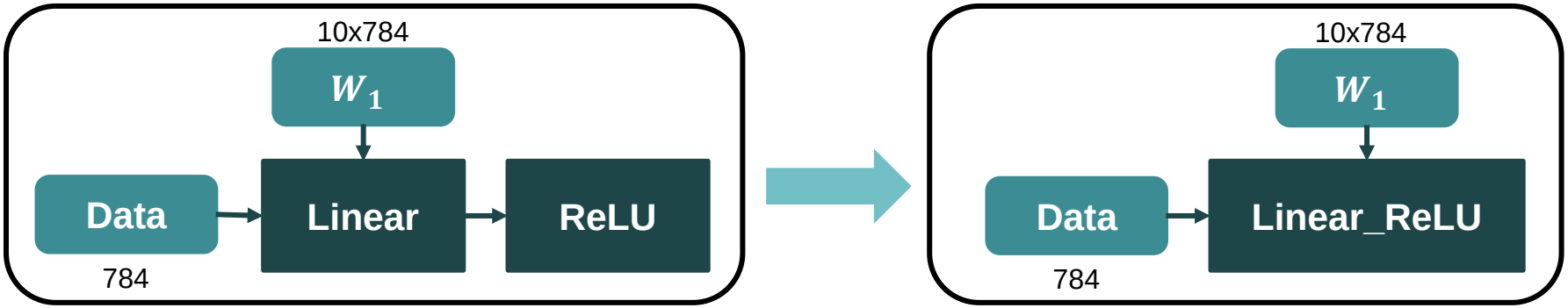


Преобразование размещения структур данных

- Для сверточных слоев поддерживается преобразование из NHWC/HWIO в NCHW/OIHW и наоборот
- При компиляции модели можно указать желаемый способ хранения данных
  - Данные конвертируются до и после выполнения свертки
- По умолчанию вместо NCHW/OIHW используется NCHWc/OIHWio
- Конвертация из NCHW/OIHW в NCHWc/OIHWio выполняется до и после каждой операции свертки
- При выборе высокого уровня оптимизации графа нейронной сети конвертация может выполняться один раз из NCHW/OIHW в NCHWc/OIHWio для нескольких последовательных сверток

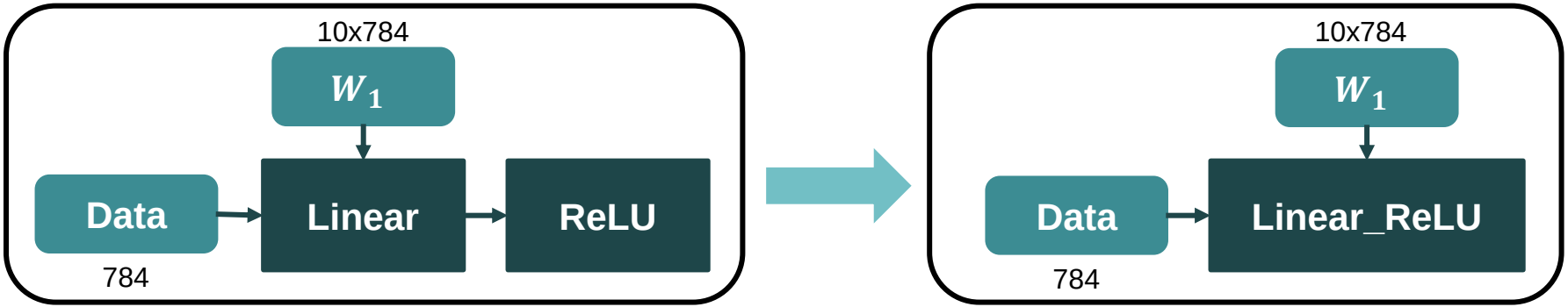
Слияние операторов…

- ***Слияние операторов (operator fusion)*** предполагает слияние нескольких преобразований в единое ядро без сохранения промежуточных результатов в памяти устройства
- Обеспечивает эффективное обращение к памяти при выполнении набора операторов
- Пример:



Слияние операторов

- Пример слияния полносвязного слоя и функции активации:




```
for i in range(I):
    for j in range(J):
        c[i] += w[i, j] * a[j]

for i in range(I):
    c[i] = max(0, c[i])
```

```
for i in range(I):
    for j in range(J):
        c[i] += w[i, j] * a[j]

    c[i] = max(0, c[i])
```

Замена операторов на более оптимизированные…

- Некоторые операторы могут быть заменены (если это возможно) на более высокопроизводительные:
  - *Вызов высокопроизводительных библиотек*
  - *Замена функций активации на аппроксимацию меньшей точности*

Замена операторов на более оптимизированные…

- На этапе оптимизации графа некоторые операторы могут быть заменены (если это возможно) на более высокопроизводительные:
  - *Вызов высокопроизводительных библиотек*
  - *Замена функций активации на аппроксимацию меньшей точности*

- TVM поддерживает замену операторов на вызов реализаций из библиотек:
  - OpenBLAS, CBLAS, cuBLAS (реализация полносвязных слоев)
  - MKL (реализация полносвязных слоев)
  - DNNL (реализация полносвязных и сверточных слоев)
  - cuDNN (реализация полносвязных и сверточных слоев)
- *Примечание:* для использования требуется сборка TVM с подключением необходимых библиотек

Замена операторов на более оптимизированные

- На этапе оптимизации графа некоторые операторы могут быть заменены (если это возможно) на более высокопроизводительные:
  - *Вызов высокопроизводительных библиотек*
  - *Замена функций активации на аппроксимацию меньшей точности*

- Для задач глубокого обучения не требуется высокая точность промежуточных вычислений
- Некоторые математические функции (функции активации) могут быть заменены аппроксимацией с помощью полиномов более низкой степени по сравнению с классической реализацией функции
- Данная оптимизация может не привести к приросту производительности, так как функции активации занимают небольшой процент от общего времени

ПЕРЕХОД К ТЕНЗОРНОМУ ВНУТРЕННЕМУ ПРЕДСТАВЛЕНИЮ

Типовая схема работы Apache TVM

Тензорное выражение

- ***Тензорное выражение (tensor expression, TE)*** – описание математического выражения в терминах тензоров и операций над ними на функциональном языке
- ***Relay IR*** описывает граф вычислений и ***определяет порядок выполнения операторов***
- ***Оператор*** должен быть представлен в виде ***тензорного выражения***
- Тензорное выражение принимает входные тензоры и создает выходной тензор
- TVM содержит примитивы для работы с тензорными выражениями:
  - Объявление переменных, тензоров разных размерностей
  - Объявление и итерирование по осям редукции
  - Итерирование по тензорам

Пример тензорного выражения

- Пример сложения двух векторов:

```
from tvm import te

A = te.placeholder((128,), name="A") # пустой тензор
B = te.placeholder((128,), name="B") # пустой тензор
C = te.compute(A.shape,              # лямбда-выражение для вычисления
       lambda i: A[i] + B[i], "C")   # суммы векторов
```

- ***Для эффективного выполнения необходимо реализовать план вычислений (schedule)***
- ***План*** – набор преобразований, применяемых к тензорному выражению
- ***Важно!!!*** Реализация нового плана вычислений не требует изменения тензорного выражения

План вычислений

- ***Тензорное выражение*** описывает, ***ЧТО И КАК*** вычислять
- ***План вычислений*** описывает, ***КАК РЕАЛИЗОВАТЬ*** вычисление тензорного выражения

- ***План вычислений описывает алгоритмическую и программную оптимизацию тензорного выражения***
- Основные приемы программной оптимизации:
  - Векторизация и распараллеливание вычислений
  - Развертывание циклов
  - Объединение циклов
  - Переупорядочивание циклов
  - Разделение одного цикла или нескольких циклов на блоки

Пример плана вычислений

- Пример разделения цикла на блоки и распараллеливание внешнего цикла сложения векторов:

```
s = te.create_schedule(C.op)           # создание пустого плана вычислений
yo, yi = s[C].split(C.op.axis[0], 16) # разделение оси на линии по 16
s[C].parallel(yo)                      # распараллеливание внешнего цикла
```

- ***Важно!!!*** Разные планы для одного и того же тензорного выражения математически эквивалентны
- Пример TIR реализации сложения двух векторов:

```
for i_outer in T.parallel(8):
    for i_inner in range(16):
        cse_var_1: T.int32 = i_outer * 16 + i_inner
        C[cse_var_1] = A[cse_var_1] + B[cse_var_1]
```

План вычислений. Операторы…

- **`Split`** разделяет указанный цикл на два вложенных цикла с размером внутреннего цикла, равным фактору (**`factor`**)

```
A = te.placeholder((m,), name="A")
B = te.compute((m,), lambda i: A[i] * 2, name="B")
s = te.create_schedule(B.op)

xo, xi = s[B].split(B.op.axis[0], factor=32)
```

- **`Tile`** разбивает два цикла на блоки указанного размера

```
A = te.placeholder((m, n), name="A")
B = te.compute((m, n), lambda i, j: A[i, j], name="B")
s = te.create_schedule(B.op)

xo, yo, xi, yi = s[B].tile(B.op.axis[0], B.op.axis[1],
                           x_factor=10, y_factor=5)
```

План вычислений. Операторы…

- **`Fuse`** объединяет два последовательных цикла

```
xo, yo, xi, yi = s[B].tile(B.op.axis[0], B.op.axis[1],
                           x_factor=10, y_factor=5)

fused = s[B].fuse(xi, yi)
```

- **`Reorder`** изменяет порядок циклов при вычислении

```
xo, yo, xi, yi = s[B].tile(B.op.axis[0], B.op.axis[1],
                           x_factor=10, y_factor=5)

s[B].reorder(xi, yo, xo, yi)
```

План вычислений. Параллелизм

- Apache TVM поддерживает параллелизм с использованием потоков операционной системы и команд SIMD
- TVM позволяет указать, какой цикл необходимо распараллеливать, а в каком – выполнять векторизацию
- Пример:

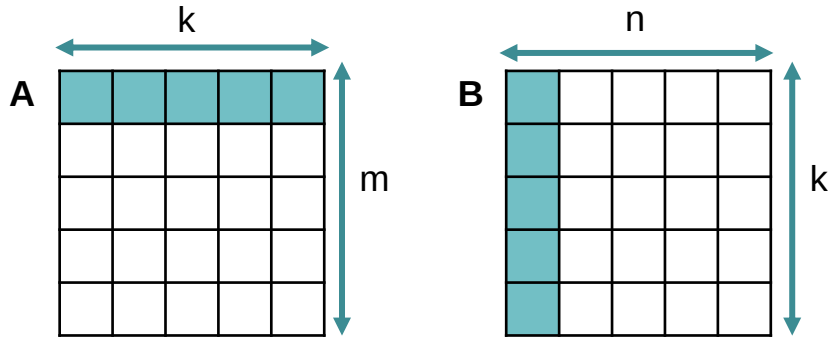
```
s = te.create_schedule(C.op)
yo, yi = s[C].split(C.op.axis[0], 16)

s[C].parallel(yo)
s[C].vectorize(yi)
```

Пример реализации умножения матриц. «Наивная» схема

- Пример умножения матрицы на матрицу:

```
k = te.reduce_axis((0, 1024), "k")           # ось редукции
A = te.placeholder((1024, 1024), name="A")   # пустой тензор
B = te.placeholder((1024, 1024), name="B")   # пустой тензор
C = te.compute((1024, 1024),                 # размер выходного тензора
      lambda m, n: te.sum(A[m, k] * B[k, n],# выражение для
                          axis=k))           # вычисления элементов
s = te.create_schedule(C.op)                 # создание объекта плана
```



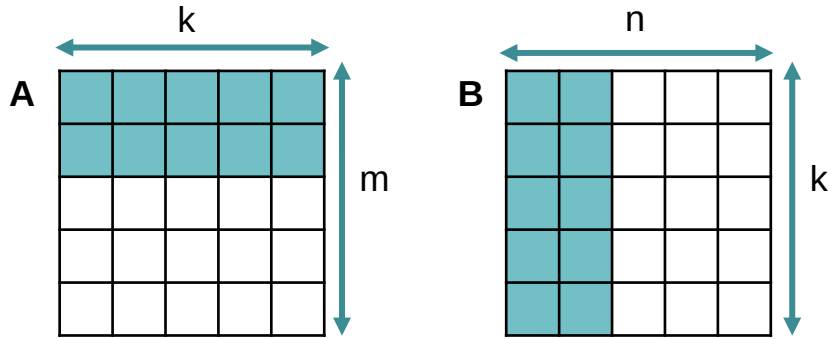
«Наивная» схема в 3 цикла

Пример реализации умножения матриц. Блочный алгоритм

- Преобразование «наивного» плана исполнения для алгоритма умножения матриц в блочный:

```
y, x = s[C].op.axis                          # размеры результирующего тензора
k = s[C].op.reduce_axis[0]                   # размер цикла редукции

yo, xo, yi, xi = s[C].tile(y, x, 8, 8) # разбиение на блоки размера 8x8, редукция
s[C].reorder(yo, xo, k, yi, xi)              # переупорядочивание циклов
```



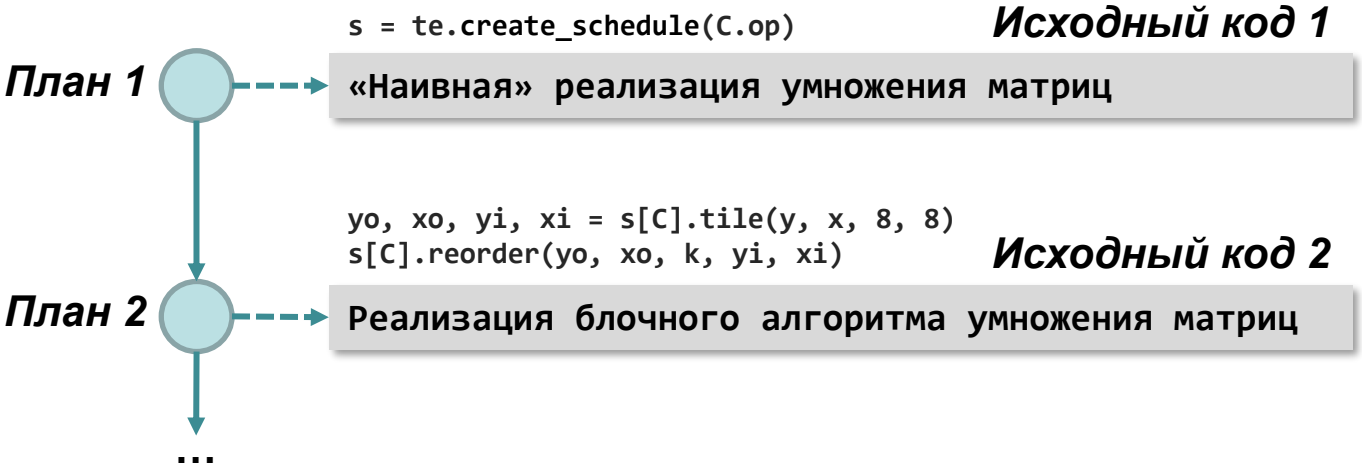
Блочная схема в 5 циклов

Пространство планов исполнения

- Применение преобразований к тензорному выражению – получение ***пространства планов***

Тензорное выражение

```
k = te.reduce_axis((0, 1024), "k")
A = te.placeholder((1024, 1024), name="A")
B = te.placeholder((1024, 1024), name="B")
C = te.compute((1024, 1024), lambda m, n: te.sum(A[m, k] * B[k, n], axis=k))
```



Оптимальный план вычислений

- В указанном примере выбран размер блока равный 8, но почему?
  - Какой размер блока выбрать?
  - Какой порядок циклов выбрать, объединить и развернуть ли внутренние циклы?
  - Как определить размер блока для нового устройства?

- TVM содержит несколько методов для автоматического выбора плана выполнения тензорного выражения, оптимального с точки зрения инструментария
- *Примечания:*
  - При запуске вычислений автоматически выбранный план не всегда позволяет достигнуть лучших показателей производительности
  - Подробнее тензорные выражения, планы вычислений и автоматический выбор планов рассматривается в следующей лекции

Что видит инженер на верхнем уровне?

Этапы не видны инженеру по высокоуровневой оптимизации

Последовательность работы инженера:
1. Отдает модель конвертеру TVM
2. Получает объектный модуль
3. Запускает вывод средствами встроенного API

СБОРКА И УСТАНОВКА APACHE TVM

Сборка и установка Apache TVM

- Варианты установки:
  - Сборка Apache TVM из исходных кодов [https://tvm.apache.org/docs/install/from_source.html]

```
git clone --recursive https://github.com/apache/tvm.git; cd tvm
mkdir build && cd build
cmake -DUSE_LLVM=ON .. && make
```

  - Сборка Runtime Apache TVM из исходных кодов [https://tvm.apache.org/docs/how_to/deploy/index.html]

```
cmake .. && make runtime
```

  - Локальная установка TVM через pip-пакет (Linux, MacOS)

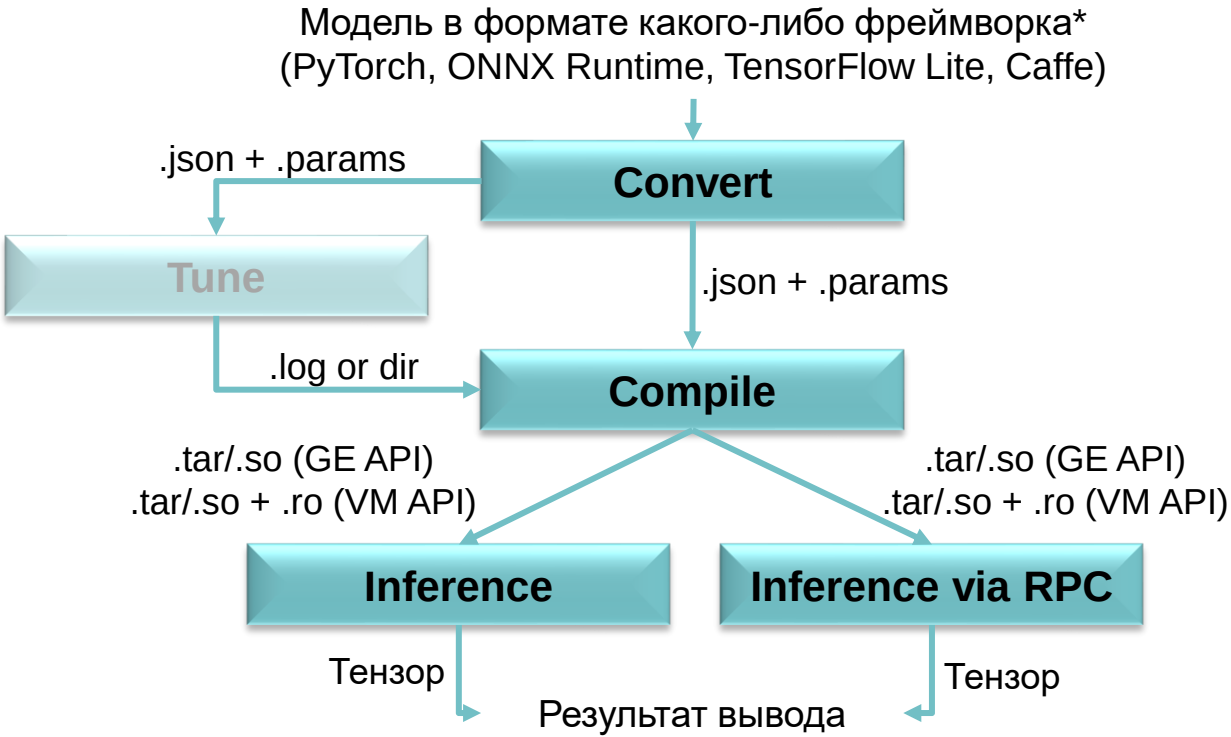
```
pip install --pre apache-tvm
```

ПРОГРАММНЫЙ ИНТЕРФЕЙС APACHE TVM

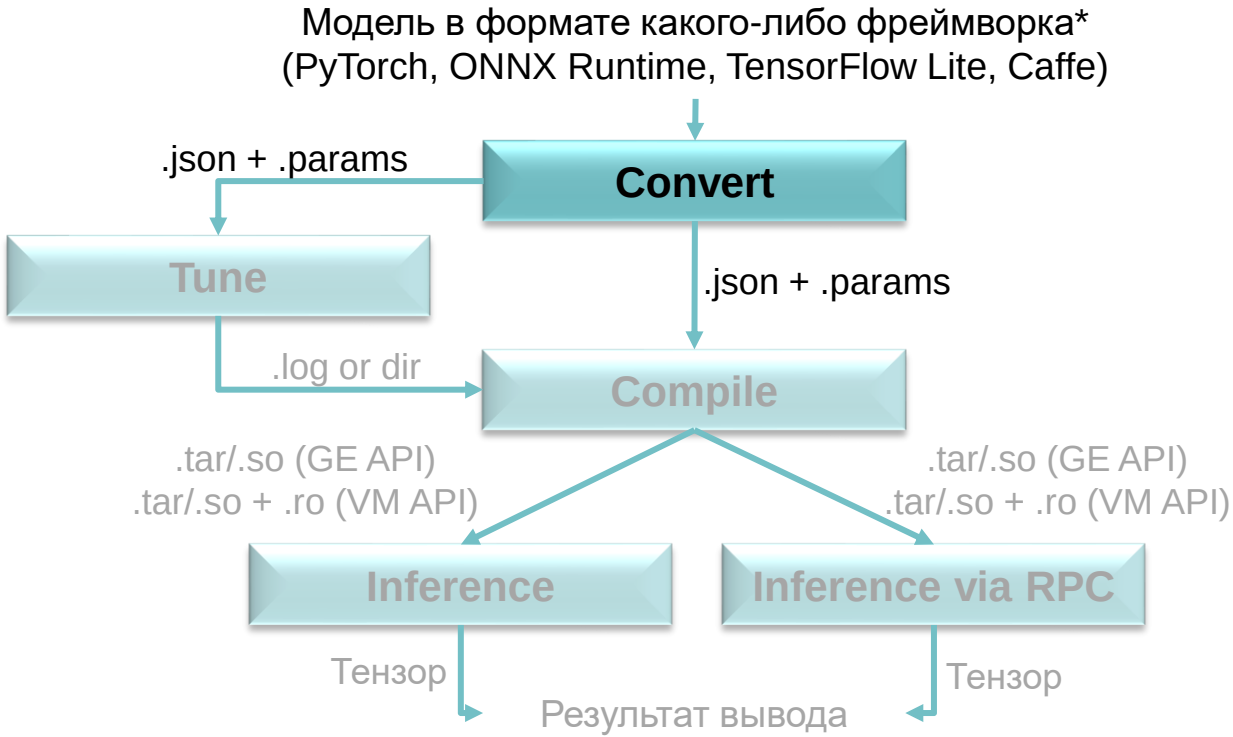
Программные интерфейсы для вывода

- Программные интерфейсы для вывода нейронных сетей:
  - **Graph Execution API (GE API)**
  - **Virtual Machine API (VM API)**
- Предоставляемые возможности:
  - Компиляция модели
  - Запуск вывода
  - Сохранение модели в формате TVM
- ***Отличие VM API от GE API*** – возможность компилировать и запускать модели, для которых на этапе компиляции *не известны размеры входных и выходных тензоров*
  - VM API используется для моделей, которые работают с любым размером входных данных (например, полностью сверточные)

Последовательность работы с моделью в Apache TVM



Последовательность работы с моделью в Apache TVM



Конвертация модели…

- PyTorch

```
input_shape = [1, 3, 224, 224]              # размер входных данных
input_data = torch.randn(input_shape)
scripted_model = torch.jit.trace(           # JIT-компиляция модели
       model, input_data).eval()            # для последующей конвертации в TVM
shape_list = [('input0', input_shape)]      # список с именем входного слоя
                                            # и размером входных данных
# конвертация модели
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)
```

- Keras

```
shape_dict = {model.input.name:             # словарь с именем входного слоя
              [1, 28, 28, 1]}               # и размером входных данных
# конвертация модели
mod, params = relay.frontend.from_keras(model, shape_dict, layout="NHWC")
```

Конвертация модели

- ONNX

```
shape_dict = {"1": [1, 1, 224, 224]}      # словарь с именем входного слоя
                                          # и размером входных данных
# конвертация модели
mod, params = relay.frontend.from_onnx(onnx_model, shape_dict)
```

- TensorFlow
  - Для конвертации рекомендуется сконвертировать модель в ONNX
  - ONNX-модель напрямую конвертируется в Relay IR (пример выше)
  - *Примечание:* TVM поддерживает прямую конвертацию из формата TensorFlow в ONNX, но она работает нестабильно из-за проблем с совместимостью с разными версиями TensorFlow

Сохранение и загрузка графа модели

- Сохранение

```
with open('param.params', "wb") as fo:
    fo.write(relay.save_param_dict(params))

with open('mod.json', "w") as fo:
    fo.write(tvm.ir.save_json(mod))
```

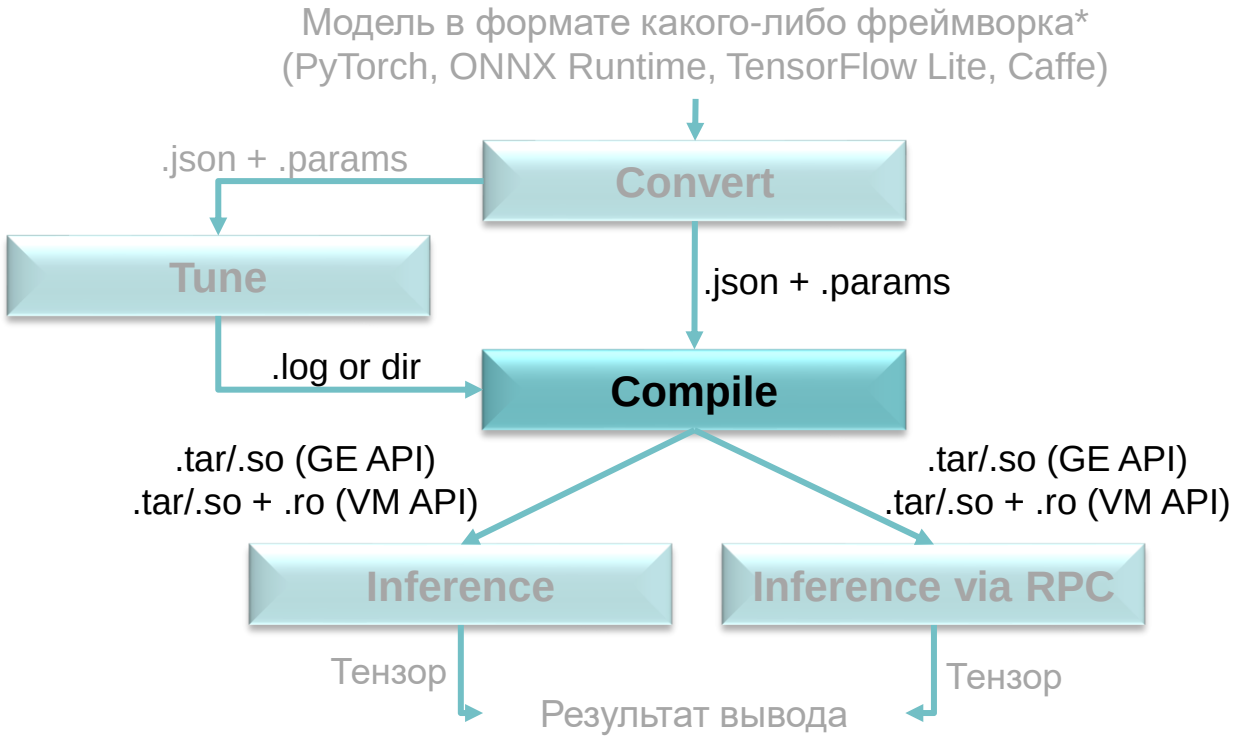
- Загрузка

```
with open('param.params', "rb") as fo:
    params = relay.load_param_dict(fo.read())

with open('mod.json', "r") as fo:
    mod = fo.read()

mod = tvm.ir.load_json(mod)
```

Последовательность работы с моделью в Apache TVM



Уровни оптимизации графа

- **`PassContext`** позволяет установить уровень оптимизации графа

```
with tvm.transform.PassContext(opt_level=0):
```

- Каждому проходу (pass) соответствует значение уровня оптимизации
- При оптимизации графа сети используются проходы, для которых установлено значение большее или равное заданному **`opt_level`**
- Можно отключить конкретные оптимизации (опция **`disabled_pass=xx`**)
- **`opt_level=3`** – ***хороший уровень оптимизации***

- *Примечание:* **`opt_level=4`** из-за слишком агрессивных оптимизаций может привести к ошибкам компиляции модели и некорректным результатам

Компиляция модели…

- Graph Execution API

```
target = tvm.target.Target("llvm")                  # строка компиляции
with tvm.transform.PassContext(opt_level=3):        # уровень оптимизации графа
    lib = relay.build(
       mod, target=target, params=params            # компиляция модели
    )
```

- Virtual Machine API

```
target = tvm.target.Target("llvm")                   # строка компиляции
with tvm.transform.PassContext(opt_level=3):         # уровень оптимизации графа
    executable = rly_vm.compile(
       mod, target=target, params=params             # компиляция модели
    )
```

Компиляция модели

- Параметр **`target`** определяет спецификацию устройства, под которое необходимо скомпилировать модель глубокого обучения
- Пример компиляции модели с использованием векторных инструкций AVX512:

```
llvm -mcpu=skylake-avx512
```

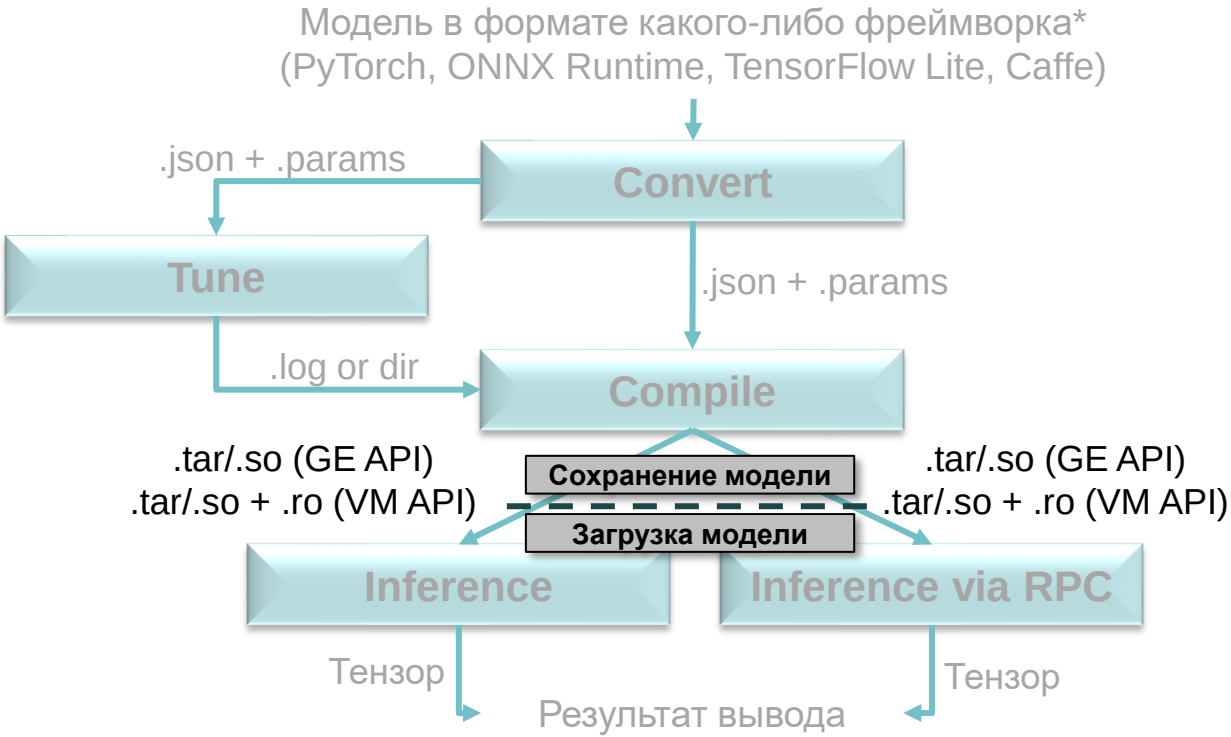
- *Примечания:*
  - По умолчанию TVM использует векторизацию SSE
  - Можно выполнять кросс-компиляцию модели. Пример строки кросс-компиляции под RISC-V:

```
llvm -mtriple=riscv64-unknown-linux-gnu -mcpu=generic-rv64 -mabi=lp64d
-mattr=+64bit,+m,+a,+f,+d,+c
```

Установка количества и привязка потоков

- Переменная окружения **`TVM_NUM_THREADS`** позволяет установить количество используемых ядер
- TVM использует привязку потоков по умолчанию, выполняя вычисления только на физических ядрах
- Изменить порядок привязки потоков можно с помощью переменной окружения **`TVM_BIND_THREADS`**
  - 0 – не использовать привязку потоков
  - 1 – привязать потоки к физическим ядрам
- Привязать потоки можно с помощью утилиты **`numactl`**, предварительно отключив привязку внутри TVM

Последовательность работы с моделью в Apache TVM



Сохранение и загрузка скомпилированной модели

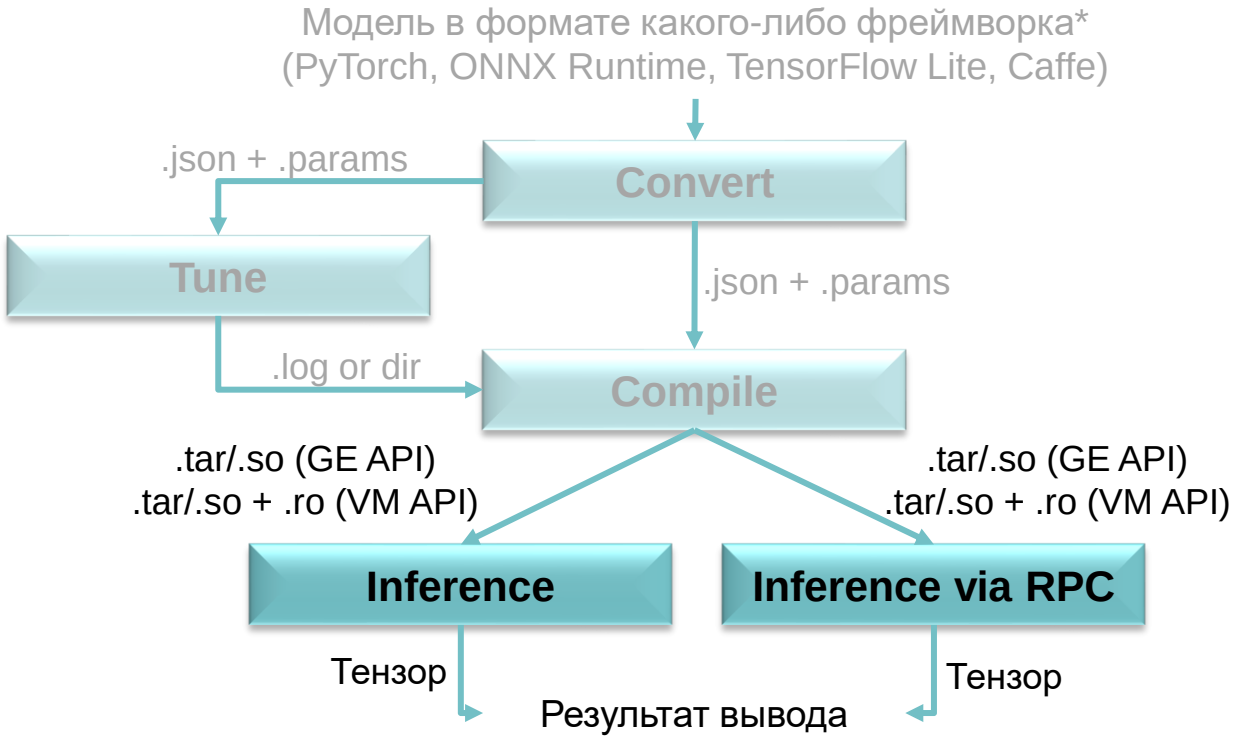
- Graph Execution API (GE API)

```
# сохранение
lib.export_library("lib.so")
lib.export_library("lib.tar")
# загрузка
lib = tvm.runtime.load_module("lib.so")
lib = tvm.runtime.load_module("lib.tar")
```

- Virtual Machine API (VM API)

```
# сохранение
code, lib = executable.save()
with open('model.ro', 'wb') as fo:
    fo.write(code)
# загрузка
code = bytearray(open('model.ro', 'rb').read())
executable = tvm.runtime.vm.Executable.load_exec(code, lib)
```

Последовательность работы с моделью в Apache TVM



Вывод нейронной сети на узле

- Graph Execution API (GE API)

```
dev = tvm.cpu(0)                                    # устройство для запуска
# создание исполняемого модуля для устройства
module = graph_executor.GraphModule(lib["default"](dev))
module.set_input(input0', img_data)                 # установка входных данных
module.run()                                        # вывод
prediction = module.get_output(0).numpy()[0]        # получение предсказаний
```

- Virtual Machine API (VM API)

```
dev = tvm.cpu(0)                                    # устройство для запуска
# создание исполняемого модуля для устройства
des_vm = vm.VirtualMachine(executable, dev)
data = {input0': img_data}                          # установка входных данных
des_vm.set_input('main', **data)                    # установка входных данных
prediction = des_vm.run()                           # вывод
```

Вывод через удаленный вызов процедур

- На устройстве запускается RPC-сервер

```
python -m tvm.exec.rpc_server --host 0.0.0.0 --port=9090
```

- Пример вывода через удаленный вызов процедур для GE API:

```
remote = rpc.connect(host, port)          # установка соединения

remote.upload("lib.tar")                  # передача исполняемого модуля
lib = remote.load_module("lib.tar")       # загрузка на устройстве модуля
dev = remote.cpu()
module = graph_executor.GraphModule(lib["default"](dev))

module.set_input('input', img)
module.run()
prediction = module.get_output(0).numpy()
```

Полный пример

- Обучение и сохранение модели в формате TVM:
  lectures/sources/06_model_training.ipynb

- Вывод модели с использованием Apache TVM (GE API):
  lectures/sources/06_TVM_GE_API_inference.ipynb

- Вывод модели с использованием Apache TVM (VM API):
  lectures/sources/06_TVM_VM_API_inference.ipynb

MICROTVM

microTVM

- **microTVM** – это часть экосистемы Apache TVM для разработки и развертывания моделей машинного обучения на микроконтроллерах и других устройствах с ограниченными ресурсами
  - Компиляция моделей машинного обучения под микроконтроллеры и встроенные платформы
  - Автоматическая настройка и оптимизация моделей на целевых устройствах по аналогии с TVM
  - Упрощение загрузки и выполнения скомпилированных моделей на целевых устройствах через разработанный программный интерфейс

Заключение

- Apache TVM реализует большой набор разных подходов к алгоритмической и программной оптимизации вывода нейронных сетей
  - Содержит высокоуровневый интерфейс для вывода
  - Поддерживает большой спектр бекэндов и вычислительных устройств
  - Выполняет оптимизацию на уровне графа вычислений и операторов нейронной сети
- По умолчанию не гарантируется наилучшая производительность, требуется проведение экспериментов для выбора оптимального режима работы
  - Параметры оптимизации графа
  - Настройка планов выполнения операторов (подбор параметров)
- ***Apache TVM активно развивается!*** Можно ожидать, что с каждой следующей версией оптимизации будут применяться более рациональным образом

Литература

1. Chen T., et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning // 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). – 2018. – P. 578-594.
2. Feng S., et al. Tensorir: An abstraction for automatic tensorized program optimization // Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 2. – 2023. – P. 804-817.
3. Roesch J., et al. Relay: A new ir for machine learning frameworks // Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages. – 2018. – P. 58-68.
4. Lai R., et al. Relax: Composable Abstractions for End-to-End Dynamic Machine Learning // arXiv preprint arXiv:2311.02103. – 2023.

Авторский коллектив

- Мееров Иосиф Борисович, к.т.н., доцент, зав. каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского
- Кустикова Валентина Дмитриевна, к.т.н., доцент каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского
- Родимков Юрий Александрович, младший научный сотрудник каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского
- Сысоев Александр Владимирович, к.т.н., доцент каф. ВВиСП Института ИТММ ННГУ им. Н.И. Лобачевского